DBMind: A Self-Driving Platform in openGauss

Xuanhe Zhou[§*], Lianyuan Jin[§*], Ji Sun[§], Xinyang Zhao[§], Xiang Yu[§], Jianhua Feng[§], Shifu Li [♣], Tianqing Wang [♣], Kun Li [♣], Luyang Liu [♣]
[§] Department of Computer Science,Tsinghua University, Beijing, China
[♣] Gauss Department, Huawei Company, Beijing, China
{zhouxuan19,jinly20,sun-j16,xy-zhao20,x-yu17}@mails.tsinghua.edu.cn,fengjh@tsinghua.edu.cn,
{lishifu,wangtianqing2,likun75,liuluyang2}@huawei.com

ABSTRACT

We demonstrate a self-driving system DBMind, which provides three autonomous capabilities in database, including self-monitoring, self-diagnosis and self-optimization. First, self-monitoring judiciously collects database metrics and detects anomalies (e.g., slow queries and IO contention), which can profile database status while only slightly affecting system performance (<5%). Then, self-diagnosis utilizes an LSTM model to analyze the root causes of the anomalies and automatically detect root causes from a pre-defined failure hierarchy. Next, self-optimization automatically optimizes the database performance using learning-based techniques, including deep reinforcement learning based knob tuning, reinforcement learning based index selection, and encoder-decoder based view selection. We have implemented DBMind in an open source database openGauss and demonstrated real scenarios.

PVLDB Reference Format:
Xuanhe Zhou, Lianyuan Jin, Ji Sun, Xinyang Zhao, Xiang Yu, Jianhua Feng, Shifu Li, Tianqing Wang, Kun Li, Luyang Liu. DBMind: A Self-Driving Platform in openGauss. PVLDB, 14(12): 2743 - 2746, 2021.
doi:10.14778/3476311.3476334

1 INTRODUCTION

Traditional databases rely on DBAs to diagnose and optimize the databases in order to meet the high-performance requirements. However, these manual methods cannot satisfy the requirements for rapidly growing users, data, and workloads, and thus it calls for a self-driving database management platform that automatically monitors, diagnoses and optimizes databases. For example, suppose a cloud database provider maintains 100,000 database instances and one DBA can manage 100 database instances. It requires one thousand DBAs to maintain these instances. To make the things worse, some tricky problems (e.g., disk crash) require DBAs to take hours to trace and recover the database.

Existing databases mainly have three limitations [3, 13]. First, there are hundreds of system metrics, and current databases cannot efficiently detect anomalies (e.g., slow query, IO contention) and potential risks (e.g., insufficient disk space) with basic statistical methods. Besides, it is expensive to rely on DBAs to detect large scale statistical data, especially for cloud databases with millions of instances. Second, existing databases cannot automatically diagnose the root causes of the detected anomalies, because there are numerous highly correlated database modules and it is laborious to rely on experts to label the anomaly cases. Third, the optimization techniques (e.g., query rewrite, index suggestion) in current databases are mainly heuristics and they may find sub-optimal solutions under complex scenarios. For example, for a nested query, openGauss creates a temporary table for the uncorrelated subquery but cannot consider the optimization within the subquery.

To address these challenges, we propose learning-based techniques, build a self-driving database platform DBMind and demonstrate the following features (Figure 1).

(1) Self-monitoring monitors and collects the information of database instances. The information includes (*i*) OS resource metrics, (*ii*) database status metrics, (*iii*) log alarm metrics. It monitors each database instance and stores the collected information in a storage system on the user side or time-series databases integrated in the server side. Moreover, Self-monitoring detects anomalies from the time-series data by (*i*) identifying abnormal indicators with spectral residual algorithm [5] and (*ii*) utilizing prediction algorithms (e.g., graph neural networks [14]) to predict future risks (e.g., slow queries, resource anomaly, performance degradation, and security anomaly).

(2) Self-diagnosis trains an LSTM model to learn root causes from both normal and abnormal data. Besides, it constructs a m (which organizes the failure category-subcategory into a hierarchy) to store representative metrics and root causes. For any abnormal data, we compute an abnormal vector with the Kolmogorov-Smirnov test and match the root cause in the failure hierarchy to detect the root cause.

(3) Self-optimization proposes learning-based techniques to optimize the databases, including reinforcement learning techniques for index recommendation, deep reinforcement learning techniques for knob tuning [4, 12], and encoder-decoder model for materialized view recommendation [2].

DBMind differs from existing database systems in two main aspects: (1) DBMind designs effective learned methods to realize self-monitoring, self-diagnosis, and self-optimization; (2) DBMind is integrated into an open source database opengauss and achieves both high usability and robustness. Experiments on real datasets have verified that DBMind can quickly discover slow SQL statements, give optimization suggestions in real time, save DBA time by over 80%, identifies and solves potential risks (e.g., disk crash).

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476334

[*] These authors contribute equally to this work

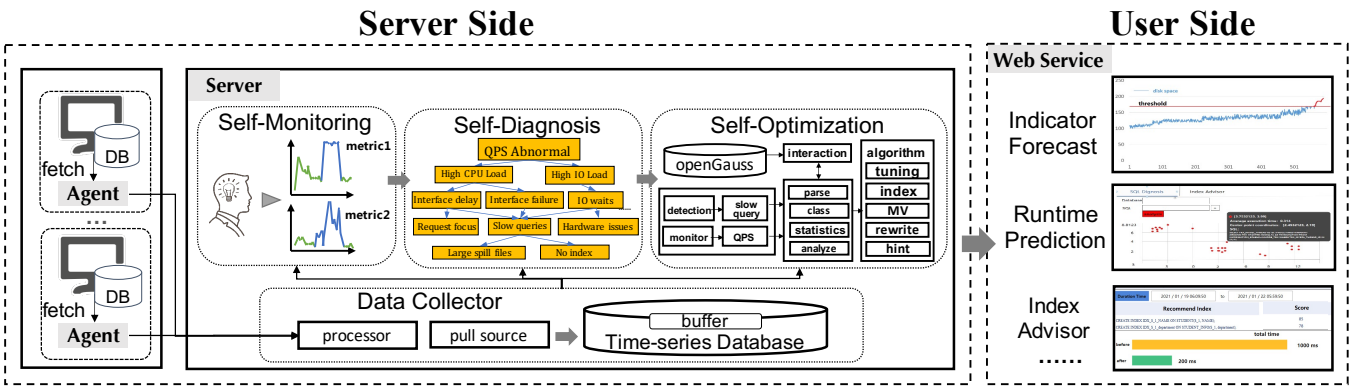

Figure 1: The Workflow of DBMind

2 SYSTEM IMPLEMENTATION

DBMind utilizes machine learning models [1, 2, 4–10, 14] to detect, analyze, and resolve database problems efficiently and effectively.

2.1 System Overview

The DBMind system is composed of three main components: `agent`, `server`, and `web interface`. Figure 1 shows the architecture of our DBMind system.

Agent Module: We first provide abundant statistic information for the users and following modules. Each agent in the database instance places three different data exporters (data collection units): *os_exporter*, *database_exporter* and *alarm_log_exporter*. (1) *os_exporter* collects the resource metrics of the operation system (e.g., CPU, memory, I/O); (2) *database_exporter* collects database metrics (e.g., *xact_commit*); (3) *alarm_log_exporter* collects the log alarm information of the database (e.g., slow queries). Some metrics of exporters are shown as the Table 1. Note that the `agent` module is extensible and we can easily add more exporters into it.

Web Service. We provide a `web interface` to help users easily monitor the running states of the database instance (e.g., resource utilization, overall health score). In the `web interface`, we provide three main functions: (*i*) demonstrate database status (e.g., resource metrics, slow SQLs, and abnormal log information) and overall health score; (*ii*) when abnormal data occurs, DBMind calls corresponding analysis and repair functions, demonstrates the optimization results (e.g., abnormality type, optimizations) and risks.

Server Module: To efficiently support the three autonomous capabilities, we devise an end-to-end optimization pipeline and abnormality detection/analysis/optimization algorithms. Section 2.2 introduces how to detect potential risks from a large amount of time-series data, and Section 2.3 describes techniques for diagnosing root causes of the detected risks. And we discuss how to judiciously optimize the database problems in Section 2.4.

2.2 Self-Monitoring Module

There are two main challenges in `self-monitoring`. First, databases generate a large amount of statistical data every day and we need to efficiently detect anomaly from those data. Second, there are noises in the logs (e.g., incompletely logged workload by interrupts) and it is important to predict the performance. Hence, the `self-monitoring` module periodically reads the latest data from the local time-series database, and detects anomaly events

Table 1: Example metrics in `self-monitoring`

Exporters	Metrics
OS_exporter	CPU utility
	disk space/IO utility
Database_exporter	index/table Information
	SQL Information
	buffer pool/lock information
Alarm_log_exporter	alarm log information

with the `Abnormal Discovery Service` and `Trend Analysis Service` (Figure 2).

(1) `Anomaly Discovery Service` analyzes and detects the key indicators of the database periodically (e.g., every second). Since there are numerous time-series data and we have few labeled data, we utilize the spectral residual algorithm (an unsupervised method) to conduct anomaly detection. The algorithm includes three steps: (*i*) Utilize Fourier transform to convert the history data (e.g., resources, OS interrupts) into amplitude spectrum and phase spectrum; (*ii*) Change the amplitude spectrum into logarithmic spectrum and obtain the significant part of the spectrum; (*iii*) Use the inverse Fourier transform to obtain the correlation saliency map.

This way, we can shrink the normal part and enlarge the abnormal data. And by setting the abnormal threshold based on the extreme value theory, we obtain the abnormal information (e.g., workload drop) and send it to the RCA module (Section 2.3).

(2) `Trend Analysis Service` analyzes the consumption of system resources (e.g., CPU, memory, disk space), and predicts the future consumption of resources based on the runtime prediction algorithm. For example, we model the concurrent queries as a workload graph, where the vertices are operators and edges are their relations, and apply graph embedding algorithm [14] to estimate resource consumption (e.g., runtime, memory consumption). If a resource is found to be insufficient to support incoming workloads, an alarm message is sent, which reflects the overall health state of the database and guides further optimization.

2.3 Self-Diagnosis Module

The Root Cause Analysis (RCA) Module accepts standardized anomaly alarms from the `Self-monitoring` module, and analyzes the root cause of the anomaly. Existing databases usually employ experienced DBA to diagnose the root causes manually, which is difficult since the state metrics are complex and lack of domain

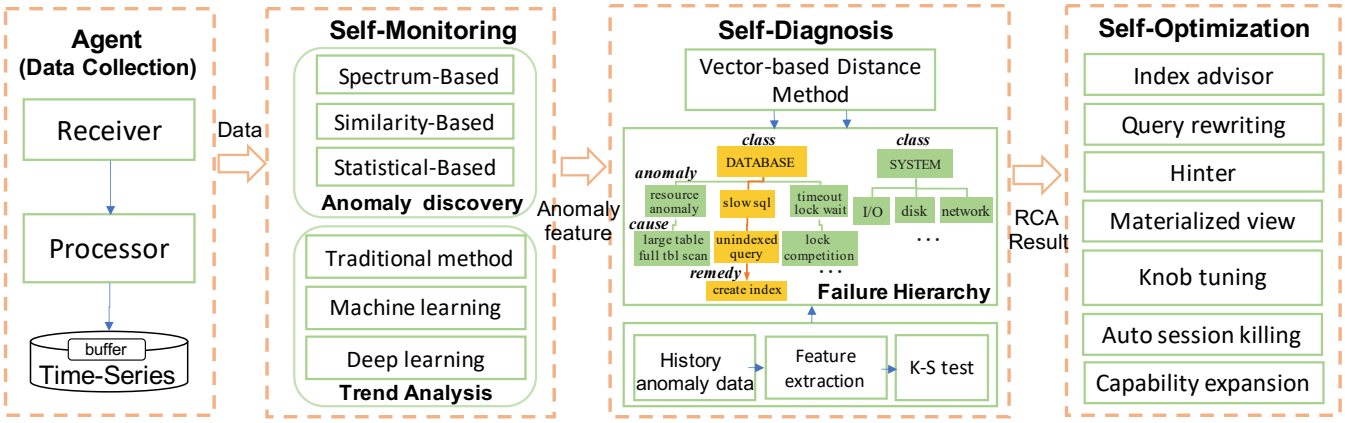

Figure 2: Learning-based Algorithms in DBMind.

knowledge. Besides, existing automatic diagnosis methods rely on expert knowledge heavily or introduce non-negligible extra cost. To address this problem, we propose a system-level diagnosis method, which builds a failure hierarchy for all anomalies where a tree node is a failure category/sub-category, regards diagnosis as a classification problem (classifying an anomaly to a failure category) and finds root causes based on the metrics data. Moreover, it optimizes the diagnosis model by analyzing historical data so that experts just label a few anomaly cases. The framework consists of two stages.

Offline Stage. We collect two types of metrics: normal data and abnormal data. First, we utilize normal data to train an LSTM-based auto-encoder model, which helps to locate the abnormal areas. Second, the abnormal data consist of database metrics, anomaly cases and the root cause label, and so we extract representative metrics based on the abnormal data and store them as a knowledge base (the failure hierarchy). For any abnormal data, we use the LSTM model to detect the abnormal area and apply Kolmogorov-Smirnov test to encode the data into an anomaly vector and match corresponding root cause in the failure hierarchy.

Online Stage. Once there is an anomaly detected by the `self-monitoring` module, `DBMind` calls the `self-diagnosis` module. It first utilizes Kolmogorov-Smirnov test on each database metric to generate the anomaly vector; and then finds similar anomaly cases and locates the root cause from the failure hierarchy.

2.4 Self-Optimization Module

The `Self-optimization` module aims to optimize the database automatically. It accepts the diagnosis result from `RCA`, enables corresponding optimization methods. `Self-optimization` mainly supports three optimization scenarios: SQL optimization, database tuning, and operation and maintenance management tuning: (i) `SQL optimization` involves learned query rewrite, index advisor, materialized view advisor. (ii) `Database tuning` involves automatic knob tuning, session killing, connection flow restriction. (iii) `Operation and maintenance tuning` involves automatic disk expansion, backup and recovery. Here we showcase some methods using different optimization algorithms.

Learned query rewriter. A slow SQL query (due to redundant or inefficient operations) can be speeded up by orders of magnitude with an appropriate rewrite order. However, it is prohibitively expensive to enumerate the orders. Hence, `Query Rewriter` inputs a query, uses *Monte Carlo tree search* to find a near-optimal rewrite order, and rewrites into an equivalent yet more efficient query.

Learned query advisors. It automatically optimizes the databases [2, 11], e.g., index/view advisor, for a workload. `Learned Index Advisor` uses deep reinforcement learning to automatically recommends indexes. `Learned View Advisor` proposes an encoder-decoder model to recommend views.

Learned knob tuner adopts a deep reinforcement learning technique to tune the knobs [4]. We use an actor-critic model to automatically select appropriate knob values, including SQL-level, session-level and system-level knob tuning.

3 DEMONSTRATION OVERVIEW

Next we showcase the interfaces of `DBMind` (e.g., the monitoring dashboard, anomaly events, optimization actions). Note that we will release more self-driving functions for public access [1].

Datasets. openGauss has been applied in banking, insurance, e-commerce fields. To verify the effectiveness, `DBMind` was tested in the bank's core transaction business scenarios. This service's data volume is about 2TB, and the peak concurrency is about 10K QPS.

End-to-End Experience. Figure 3 is a screenshot of the front-end of `DBMind`. The user can observe database activities and health states and pose optimization requests with the following steps.

1) Monitor Database Performance. Users can browse the monitoring metrics (e.g., CPU usage, memory usage) on the web page. Meanwhile, `DBMind` scores the database based on the occurring rate of potential risks (Figure 3-①).

2) Detect Root Causes. `DBMind` automatically monitors the abnormal status of the database instance and highlights the abnormal points on the curve (Figure 3-②). The user can interactively check the results. She puts the cursor onto the curve, and a text box will occur to show the root cause of the event, i.e., an I/O intensive event caused the workload performance abruptly decreased (Figure 3-②).

3) Check Self-Optimization History. Besides the root causes, users can check the repair behaviors in the selected time period. As shown in Figure 3-③, the self-optimization history panel displays the past repair behaviors. Users can check these behaviors. And if they think some behavior is incorrect (e.g., unnecessary index), they can undo the behavior by clicking the "*withdraw*" button. And if they want to

[1] http://admin.dbmind.cn

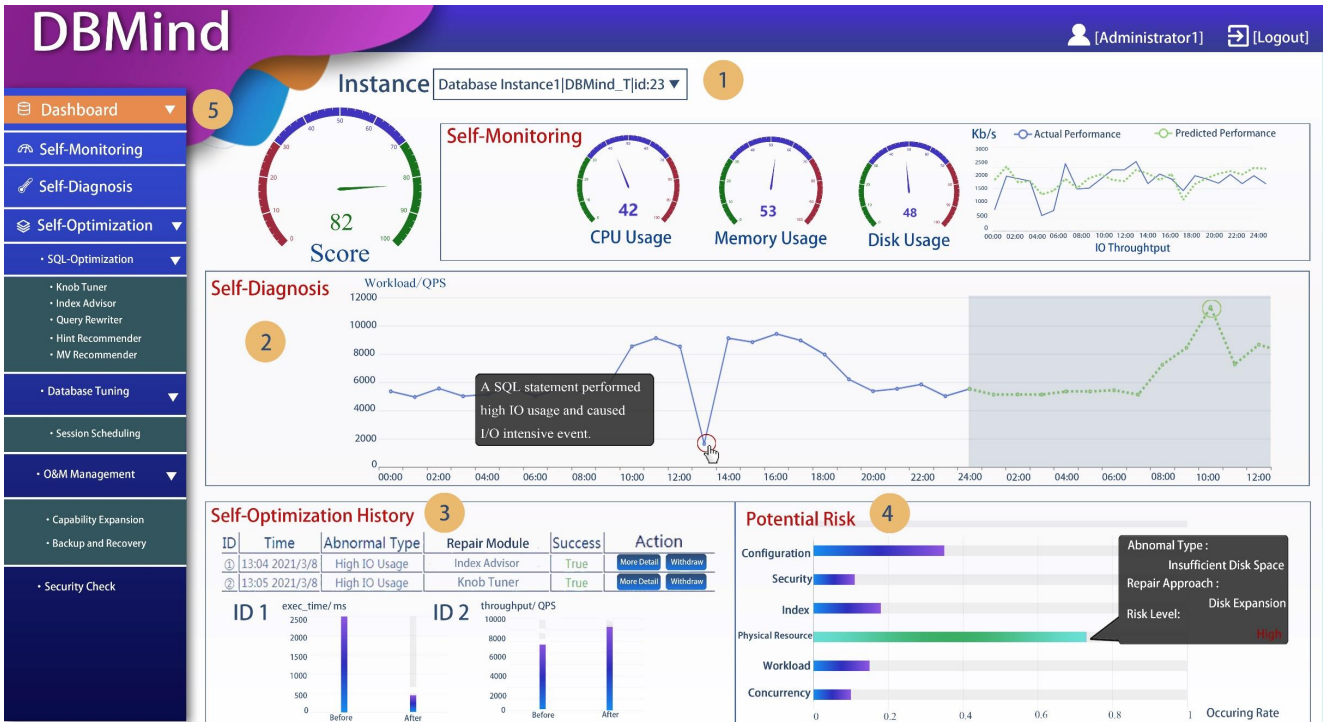

Figure 3: A Screenshot of DBMind

learn more about the created index, they can click the "*more details*" button and it will jump to the *Index Advisor* page.

4) Check Potential Risks. Moreover, users can check the potential risks that DBMind proactively found, like configuration/workload-/resource problems. In Figure 3-④, DBMind identifies the problems that may occur in future and some non-emergency risks (e.g., insufficient disk space). With the risk, users can reassess the employment.

5) Utilize Database Optimization Tools. DBMind offers many useful `self-optimization` tools. To simplify utilization, there are two access ways. (*i*) Users directly click the "*more details*" buttons in the history panel (Figure 3-④) and jump to the corresponding tool page; (*ii*) Users click the corresponding tool button in Figure 3-⑤.

Scenario 1 - Slow SQL Diagnosis.

As shown in Figure 3, if DBMind found that the throughput suddenly dropped (`Self-monitoring`), the RCA module received the anomaly, matched the anomaly in the failure hierarchy, and found the root cause, i.e., *lack of index*. Hence, we enable *index advisor* function. For the whole workload, we vectorize query features and use reinforcement learning to recommend a set of indexes. Besides, for some extremely slow queries, we analyze the semantic information of the query statement and gather the statistical information of the database. Then, we calculate the selectivity of the predicates and recommend indexes based on the high cost predicates.

Scenario 2 - Knob Tuning.

As shown in Figure 3, since *high I/O usage* can also be caused by improper knob values, DBMind conducts *knob tuning* in workload level, including offline tuning and online tuning. For offline tuning, users can trigger the tuning function by selecting the `Knob Tuner` function, which occupies a certain amount of system resources for model training and testing. Note that, while training the model, a user-specified benchmark is repeatedly executed on a synchronous database to feed back the reward. For online tuning, DBMind recommends knobs based on the performance metrics (extracted by `self-monitoring`). This process only takes seconds to finish. And then the user can check whether these knobs are correctly recommended and (optional) commit the adjustment (Figure 3-④). If the recommendation is unsatisfying, the use can restore the default configuration by clicking the "*withdraw*" button.

ACKNOWLEDGMENTS

This work was supported by NSF of China (61925205, 61632016), Beijing National Research Center for Information Science and Technology, Huawei, TAL education.

REFERENCES

[1] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.
[2] Y. Han, G. Li, H. Yuan, and J. Sun. An autonomous materialized view management system with deep reinforcement learning. In *ICDE*, 2021.
[3] G. Li and et al. AI meets database: AI4DB and DB4AI. In *SIGMOD*, 2021.
[4] G. Li, X. Zhou, B. Gao, and S. Li. Qtune: A query-aware database tuning system with deep reinforcement learning. In *VLDB*, 2019.
[5] G. Li, X. Zhou, S. Ji, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *VLDB*, 2021.
[6] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *Data Eng. Bull.*, 2019.
[7] M. Li, H. Wang, and J. Li. Mining conditional functional dependency rules on big data. *Big Data Mining and Analytics*, 03(01):68, 2020.
[8] J. Sun and G. Li. An end-to-end learning-based cost estimator. *VLDB*, 2019.
[9] S. Tian, S. Mo, L. Wang, and et al. Deep reinforcement learning-based approach to tackle topic-aware influence maximization. *Data Science and Engineering*, 2020.
[10] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308, 2020.
[11] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512. IEEE, 2020.
[12] J. Zhang, Y. Liu, K. Zhou, G. Li, and et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, 2019.
[13] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 2020.
[14] X. Zhou, J. Sun, G. Li, and et al. Query performance prediction for concurrent queries using graph embedding. *VLDB*, 2020.